1 Lab 09: Distributed Sensing

This laboratory requires the following:

- C development tools (gcc, make, etc.)
- Matlab
- Webots simulation software
- Webots User Guide
- Webots Reference Manual
- TinyOS development suite, version 2.1.x
- 1 USB to nine-pin serial cable
- 1 MICAz log and communication board
- 2 AA batteries
- 1 MIB510 programmer/interface board
- 1 MTS300CA sensor board

Depending on your programming skills, the laboratory duration might require an effort of up to five hours although assistance will be provided in the computer rooms during a time window of three hours. Although this laboratory is not graded, we encourage you to take notes during the course of this laboratory to aid in preparing the final exam. A solution to this lab will be posted after the lab session.

1.1 Office hours

Additional assistance outside the lab period (office hours) can be requested using the dis-ta@groupes.epfl.ch mailing list.

1.2 Information

In the following text you will find several exercises and questions.

1. The notation $\mathbf{S_x}$ means that the question can be solved using only additional simulation or an experimental manipulation.
2. The notation $\mathbf{Q_x}$ means that the question can be answered theoretically, without any simulation or experimental manipulation.
3. The notation $\mathbf{I_x}$ means that the problem has to be solved by implementing a piece of code and performing a simulation or experimental manipulation.
4. The notation $\mathbf{B_x}$ means that the question is optional and should be answered if you have enough time at your disposal.

To prepare yourself for the exam and to allow you for better time planning during the exercise session, we show an indicative number of points for each exercise between parentheses. The combined total number of points for the laboratory or homework exercises is 100.

2 Getting familiar with sensor nodes, The MICAz log and communication board

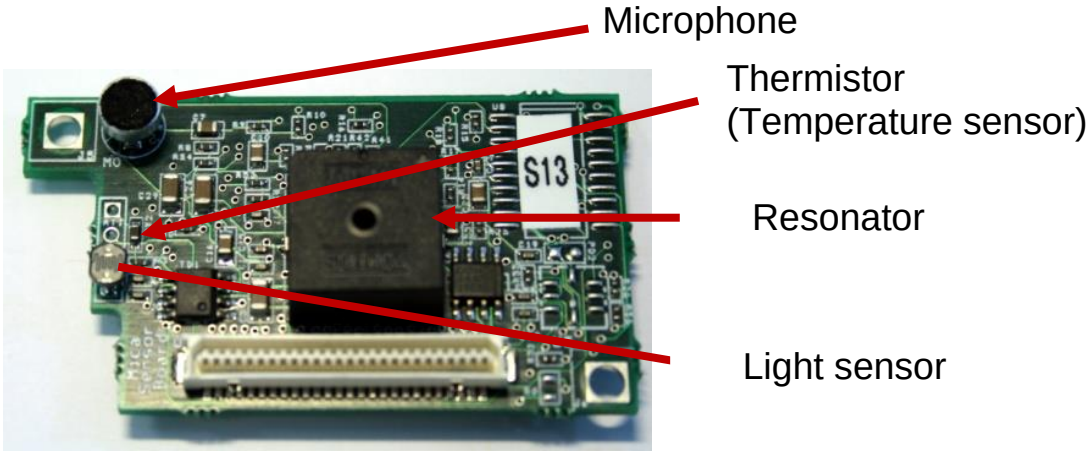
The MICAz is a 2.4GHz (IEEE 802.15.4 compliant) module used for enabling low-power wireless sensor networks. Its features include:

- IEEE 802.15.4/ZigBee compliant RF transceiver
- 2.4 to 2.4835 GHz, a globally compatible ISM band
- Direct sequence spread spectrum radio which is resistant to RF interference and provides inherent data security
- 250 kbps data rate
- An Atmel ATmega128L microcontroller running TinyOS 2.1.x
- A modular design allowing for extension with a wide range of additional modules (sensor board, data acquisition board, etc.)

2.1 The MTS 300 CA sensor board

The MTS 300 CA is a sensor module compatible with the MICAz that contains:

- Light sensor (Clairex CL94L)
- Temp sensor (Panasonic ERT-J1VR103J)
- Acoustic sensor (WM-62A Microphone)
- Sounder (4 kHz Resonator)



2.2 TinyOS

TinyOS is an open-source barebone operating system designed for wireless embedded sensor networks. It features a component-based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks. TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools – all of which can be used as-is or be further refined for a custom application. TinyOS's event-driven execution model enables fine-grained power management yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces.

All TinyOS programs are written in nesC, which is a programming language especially designed for TinyOS. The basic syntax is very similar to what you have learned in C, but with some major additions. For a complete reference and good tutorials, look at the TinyOS website (https://github.com/tinyos/tinyos-main).

Making your own TinyOS program is done in two steps:

1. Creating your own **component** (equivalent to your `main.c` file in C) which is calling TinyOS functions (contained in other components).
2. Creating a **configuration file**, whose purpose is to link components together, telling which other components you will effectively use in your final application and the way you connect them together.

In this lab we do not develop codes for MICAz. We have already provided for you the code and it is already compiled. You just need to flash the code into the MICAz and run the experiment in order to get a feeling how a sensor node works in reality.

2.3 Testing the hardware/software setup

Start by verifying that all the tools we provide you with are working properly:

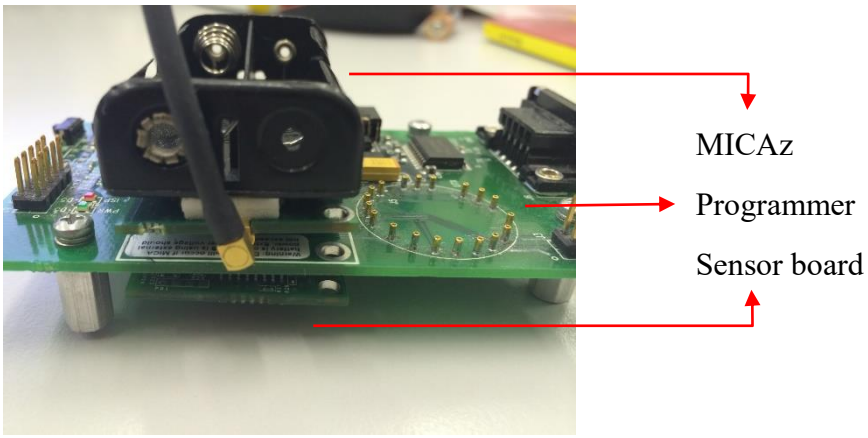
1. Extract the provided tools:

```
tar -xzf lab09.tar.gz
cd Lab09
```

2. Check that your environment properly references a working TinyOS tree on this machine by executing "`tos-check-env`". You should see a lot of output, followed by a list of errors. You can safely ignore errors relating to your CLASSPATH not including '.', your version of Java or GraphViz. If you see other errors please ask a TA, or try a different machine.

3. Plug the MIB510 programmer board into the computer using the USB-to-serial cable. Make sure that you plug the USB side into the USB connection on the PC and **not** on the monitor. **Make sure that the switch on the programmer board is in the OFF position**. Attach a MICAz mote to it and a sensor board on the opposite side of the programmer (see the picture below). Make sure that the connector is well plugged in. Then, flip the switch on the mote to ON. The

programmer will remain OFF for the entire lab. **Do not forget to insert batteries into your MICAz mote.**



2.4 Sensing light with the MICAz

We are going to use the MICAz to sense the ambient light in the room and send it back through the serial interface to the computer, where it will be logged in a text file.

First, we need to program the MICAz with our sensing application. Go to the right directory:

```
cd TinyOS_code/ build/micaz
```

Then, to get access to the comm port of the computer, you need to run the following command:

```
ssh localhost
```

Enter your GASPAR password if you are asked.

Then, **switch on the MICAz** and flash the mote (all in one line):

```
uisp -dprog=mib510 -dserial=/dev/ttyUSB0 --
wr_fuse_h=0xd1 -dpart=ATmega128 --wr_fuse_e=ff -
-erase --upload if=main.srec –verify
```

If you see "`flash error at address 0xXXX: file=0xXX, mem=0xXX`" while programming the MICAz, just ignore it.

After the node has been successfully programmed, turn off and on the MICAz. You must see the LEDs c9 c10 and c11 on the programmer board blinking. If you see the blinking LEDs, turn off the MICAz and go to the next steps.

Now we need to start a program that listens to the serial port so that we can then display the incoming messages to the console or write them to a log file.

Go to the folder `Lab09/Matlab_codes_for_micaz` and launch the following commands in this order (be aware that copy & paste the commands may result in errors, it is recommended to type them instead):

-
  ```
  ssh localhost
  ```

- in one single line:

  ```
  java net.tinyos.tools.Listen -comm
                                serial@/dev/ttyUSB0:micaz
  ```

  → If you see errors related to '`toscomm`' just ignore it. You should see some data showing up in your screen. If you do not see running data, make sure the MICAz is on and the LEDs are blinking.
- Now that you saw the data is coming, press ctrl+z to stop the command above. Run the following command (in one single line) to save data into a log file:
  ```
  java net.tinyos.tools.Listen -comm
                      serial@/dev/ttyUSB0:micaz > log.txt
  ```
  where `log.txt` is the name of your log file. If you see errors related to '`toscomm`', ignore it.

S1 (2): For this experiment you should leave the light sensor facing down and try to keep the lighting conditions as stable as possible. Use a stopwatch and start it at the same time that you **turn on the MICAz**. After 20 seconds stop the logging program (the last command you ran above) by pressing ctrl+z.

Open the log file, and see if there is data in it. If there is no data, there has been something wrong in the process, you may need to repeat the experiment and turn off and on the MICAz.

Q2 (3): Open Matlab. Set the current folder to `Matlab_codes_for_micaz` and run the file `analyzeSensorData.m`:

```
analyzeSensorData('log.txt');
```

How does the plot of the signal look like? Why do you think there are variations in the signal and is not completely constant?

S3 (3): Run a second experiment like the one before, but, this time, try to change the lighting conditions (by casting a shadow on the sensor or by moving the node repeatedly). Have a look at the data collected using the same Matlab function as before.

What do you observe?

Q4 (3): Open file `SenseC.nc` (in `TinyOS_code` folder), read the code and try to understand how the node works. Find the line where the sampling interval is defined.

Q5 (3): Go to file `analyzeSensorData.m` and explain how the code is working. Where the sampling period is defined?

3 Energy Efficiency in Distributed Sensing

Efficient use of limited energy resources is a central theme of the field of distributed sensing. In many applications, sensor networks need to be able to operate over long periods of time with limited or no external intervention after the initial deployment. For this reason much effort has been placed in developing intelligent algorithms capable of maximizing the operation time of distributed sensing systems through node activity management.

In order to evaluate the performance of a distributed monitoring system an adequate metric is needed that can integrate the tradeoff between field estimation quality and energy cost. A general performance metric for monitoring phenomena without prior assumptions about the form of the signal or type of application can be expressed by:

$$M_C(\alpha,\beta,\gamma,\delta) = \alpha \cdot \left(1 - \frac{1}{\varphi_{max} - \varphi_{min}} \cdot \sqrt{\frac{\sum_{n=1}^{N}\left(\hat{\varphi}_n(x,y,t) - \varphi_n(x,y,t)\right)^2}{N}}\right) + \beta \cdot \left(1 - \frac{\sum_{k=1}^{K} S_k}{K \cdot T \cdot F_s / L_s}\right) + \gamma \cdot \left(1 - \frac{\sum_{k=1}^{K} P_k}{K \cdot T \cdot F_m}\right) + \delta \cdot \left(1 - \frac{\sum_{k=1}^{K} V_k}{K \cdot T \cdot v_{max}}\right)$$

where:

φ_n: the fully-sampled dataset
$\hat{\varphi}_n$: the estimated dataset (based on the subsampled measurement set)
φ_{min}: the minimum observed value
φ_{max}: the maximum observed value
N : the number of samples in the fully-sampled dataset
K : the number of nodes in the network
S_k : the number of measurements taken by node n
T : length of experiment (time)
F_s : sampling frequency
L_s : samples per measurement
P_k : the number of messages sent by node n
F_m : maximum message transmission rate
V_k : length of agent n's trajectory
v_{max} : maximum agent velocity

The weights $(\alpha,\beta,\gamma,\delta)$ may be balanced according to the severity one wishes to associate with each of them, as long as they sum to one for normalization.

In this part of the lab we will focus on the specific task of environment monitoring, by considering a group of e-pucks which are sensing a light field.

4 The Static Sensor Network

In Section 2, you experienced how a sensor node works and how you can get data from it into a base station. Now we move to simulation and we will experience multiple

sensor nodes as a sensor network. Load the *static_net* world. It contains a network of 16 static e-puck robots distributed on a regular grid. Each robot is equipped with an upward facing light sensor sampling a static light field. For this scenario we consider a quad-tree network topology (see Fig. 1), with robots using short-range radio messages to communicate between each other and the highest hierarchical cluster-head providing the only long-range up-link.

Q6 (3): Compile the two controllers (the supervisor: *static_sup.c*, and the robots' controller: *static_con.c*). Note that all the robots have the same controller. Have a look into the *static_con.c* code. How is the quad-tree network topology implemented? How does a robot find its leaves in the network?

Q7 (3): Run the simulation and wait until finished. Read the output messages that is printed on the console. Note that all the messages sent to the supervisor have come from robot0. Why is this? Which robots forward some messages from other robots? Why?

S8 (2): By running the simulation, the supervisor produces a file containing all the relevant data named *output.m*, in the *Lab09/matlab/* folder. Open Matlab, import the data by running the output file and then run the *evaluate.m* script.

```
>> output
>> evaluate
```

Q9 (3): Have a look at the loaded variables. Which variable contains the sampled set? Which variable contains the reference set? What is the size of each one?

Q10 (3): Have a look at the code of *evaluate.m*. How is the performance of the sensing system computed? Are all the terms of the general metric computed? Why?

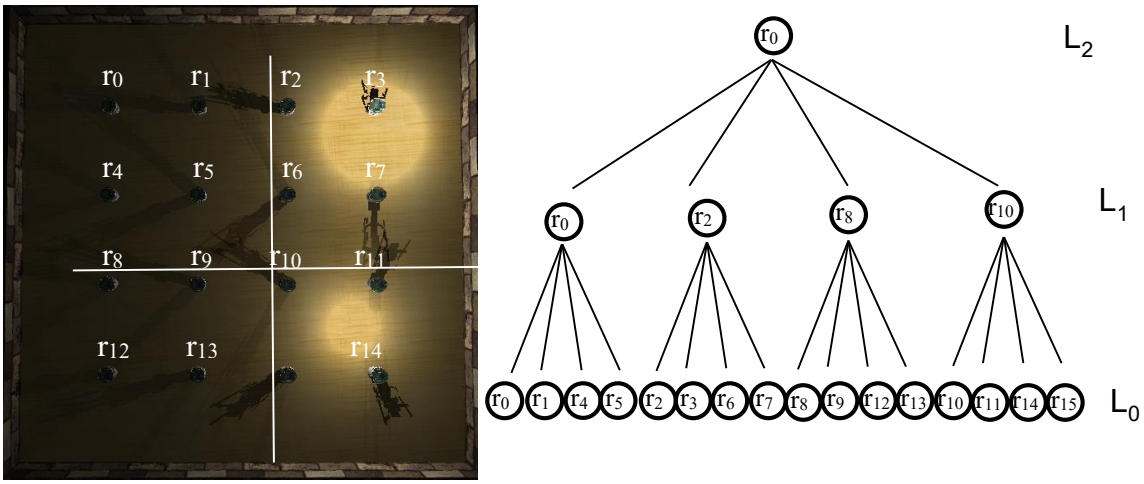


Figure 1: Quad-tree topology of static network

4.1 Spatial Suppression

Since the performance metric that we use balances field estimation quality with communication cost, one way to improve it is by reducing the number of sent messages by suppressing the transmission of measurements from homogeneous areas (with low information value), and maintaining higher resolution in areas of contrast in the underlying target field.

This can be achieved by exploiting the quad-tree network topology. The cluster-heads on each hierarchical level, which in the default controller acted solely as message relays towards the higher levels, can be modified to decide whether the information received from the other cell members contains enough information to be forwarded or should just be dropped.

I11 (10): Load the *space_division_net* world. Edit the incomplete *prun_stat_con* robot controller and implement a threshold-based management for message forwarding through cluster-heads. Compile and run it. (*Hint*: the robot_id variable is used to determine whether the robot is a cluster-head. The part of the code you need to implement is marked with TODO. You should set correct values to three variables: *sw*, *isCellApprox*, and *prunLevel,* so first read the descriptions of these variables in the code. For more information take a look at the function 'send_data' and see how the structure of a message is.)

Q12 (4): What performance do you obtain in this case? How does it compare with uniform spatial sampling method? (*Hint*: use the same Matlab scripts, *output.m* and *evaluate.m*, for evaluating the performance).

Note: The algorithm you have just implemented is a basic variant of backcasting. For a detailed description of how to optimally select the "pruning" thresholds refer to [1].

4.2 Temporal Suppression

Now we consider a direct link between each node and the base station (no tree structure), while the field (light) is dynamically changing.

S13 (1): Load the *uniform_net world*. Check the codes (the controller `uniform_con.c` and the supervisor `dynamic_field_sup.c`) and see how the network is built.

S14 (2): Compile the controllers and run the simulation and evaluate the performance of the network in Matlab using the *output.m* and *evaluate_dynamic.m* scripts.

Q15 (5): How is the field changing? Are all points of the field equally affected?

Another axis for adaptive control of a sensing process is the temporal one. Consider a time-division measurement scheduler as the one presented in Fig. 2. Assuming a model of the underlying process, the controller switches to a low sampling frequency when the acquired data matches its model and increases the sampling frequency when the measurements deviate from this model.

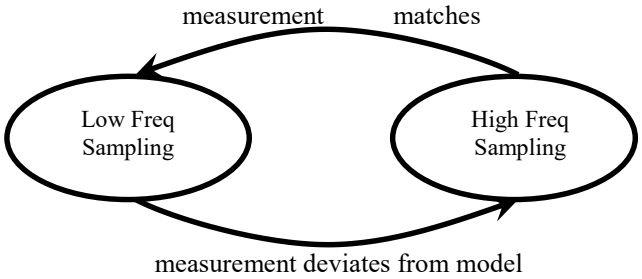


Figure 2: Basic time-division scheduler

I16 (12): Load the *time_division_net* world. Edit the *time_division_con* robot controller and implement a threshold-based algorithm for switching between high frequency and low frequency sampling, based on a linear process model. Compile and run it. (*Hint*: the part of the code you need to implement is marked with TODO. Here is one way to do it: always keep the last two sampled values (e.g., you can name them prevValues[1] and prevValues[0]) and their time stamps (e.g., named prevTstamps[1] and prevTstamps[0]). Note that there is a counter in the loop (named 'count'). With these variables (prevValues, prevTstamps, and count), using a linear model, calculate the expected value in each iteration. If the difference between the sensor reading and the expected value is less than a threshold (let's say 15), then change the Ts to have a lower sampling frequency.)

Q17 (4): What performance do you obtain in this case? How does it compare with uniform temporal sampling method? Run the world and evaluate the performance of the network in Matlab using the *output.m* and *evaluate_dynamic.m* scripts.

5 Mobile Sensor Network

Adding mobility to a sensor network presents the obvious benefits of being able to sample multiple locations with the same sensor node, increasing the coverage and spatial resolution of the sensing system. Its main drawback is the reduction in temporal resolution at any given location.

One broad classification of the different types of mobility relates to whether the measurements acquired by a sensor node play any role in shaping its trajectory. By this criterion we have either controlled or uncontrolled mobility. In the last part of the lab you will learn the impact of mobility of a sensor network on the performance of the monitoring system.

5.1 Uncontrolled Mobility

Load the *mobile_net* world, compile its default controllers and run it.

S18 (4): Evaluate the performance of the network in Matlab using the *output.m* and *evaluate_dynamic.m* scripts. How does the performance compare with the performance of the temporally-uniform sampling method?

S19 (4): Edit the dynamic_field_sup controller and increase the dynamics of the field by modifying the FIELD_DYNAMICS (by setting it to 1). How does this affect the performance of the sensing system?

Until this point we did not consider the cost of mobility in the performance metric. In some scenarios this might be a realistic assumption, as the sensor node might be attached to an otherwise independent source of mobility (e.g., a sensor node attached to a city bus, a smartphone attached to a pedestrian, etc.). This particular type of mobility is called parasitic.

S20 (2): Re-evaluate the performance obtained with the randomly moving e-pucks to take into account the cost of mobility by using the *output.m* and *evaluate_mobile.m* scripts.

5.2 Controlled Mobility

The purpose of controlled mobility is to adjust behavior of the robot based on the environmental situation, i.e. the measurements acquired by the sensors node play a role in shaping the trajectory of the robot.

Q21 (4): How would you design a controller capable of guiding the e-puck robots in order to maximize their performance of gathering the information from the field?

In this part you will implement a basic robot controller capable of guiding the e-pucks in order to maximize the information gathered from the field. The behavior of the controller should be the following:

- using its short-range radio link each robot broadcasts locally the location where it has observed the largest measurement gradient
- the velocity of the robot at each time step is updated as a weighted sum between the previous velocity, a vector pointing towards its historical best location and a vector pointing towards the neighborhood best and a random walk vector.

I22 (12): Load the *controlled_mobile_net* world and open in the editor the *guided_con.c* controller. Your task is to implement the strategy explained above in the code. (*Hint*: the part of the code you need to implement is marked with TODO. We have already provided a function named '*compute_wheel_speeds'* which calculates the wheel speeds given a *pose*, *heading* and *intended target point*.) Take a look at this function ('*compute_wheel_speeds'*) and understand what it does. Compile and run your code. Note: tuning the weights of this controller to get a high performance is not in the scope of this lab, so do notwaste much time on that.

Q23 (4): What does this algorithm remind you of?

Q24 (4): Compare the performance of the controlled mobility network with the performance of the uncontrolled mobility. Which one is better? Justify your results. Use the *output.m* and *evaluate_mobile.m* scripts.

6 References

[1] A. Prorok, C. M. Cianci and A. Martinoli. Towards Optimally Efficient Field Estimation with Threshold-Based Pruning in Real Robotic Sensor Networks. 2010 IEEE International Conference on Robotics and Automation, Anchorage, Alaska, USA, IEEE International Conference on Robotics and Automation ICRA, 2010.